Faster Algorithms for Computing All Prime Implicants of a Boolean Expression

AG1

RUBAB Tamzid Morshed

HKUST

May 4, 2023

advised by
Amir GOHARSHADY

Table of Contents

Context

The initial goal of this project was to find faster algorithms for computing Grobner basis of a system of polynomial equations. However, as I studied more about this problem, I realized it was out of scope for this thesis. So, I found a special case of the problem to work on. That is the problem of computing all prime implicants of a boolean expression[1].

[1] Please see my final report for the precise mathematical connection

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b
- A prime implicant is a "minimal" implicant (i.e., no subset of it is an implicant)

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b
- A prime implicant is a "minimal" implicant (i.e., no subset of it is an implicant)
- Helps simplifying/minimizing boolean expression, which is a common problem in CS

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b
- A prime implicant is a "minimal" implicant (i.e., no subset of it is an implicant)
- Helps simplifying/minimizing boolean expression, which is a common problem in CS
- Quine-McCluskey's algorithm does it by first computing all prime implicants of the expression and then choosing the essential ones

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b
- A prime implicant is a "minimal" implicant (i.e., no subset of it is an implicant)
- Helps simplifying/minimizing boolean expression, which is a common problem in CS
- Quine-McCluskey's algorithm does it by first computing all prime implicants of the expression and then choosing the essential ones
- Computing all prime implicants of a boolean expression is an np-hard problem

Motivation (Prime Implicant)

- An implicant of boolean expression b is a conjunction of boolean literals (e.g., $x_1 \wedge \neg x_2$) that implies b
- A prime implicant is a "minimal" implicant (i.e., no subset of it is an implicant)
- Helps simplifying/minimizing boolean expression, which is a common problem in CS
- Quine-McCluskey's algorithm does it by first computing all prime implicants of the expression and then choosing the essential ones
- Computing all prime implicants of a boolean expression is an np-hard problem
- We are interested in parameterized algortihms for this problem

Motivation (Parameterized Algorithms)

- Faster algorithms for np-hard problems

Motivation (Parameterized Algorithms)

- Faster algorithms for np-hard problems
- Assuming some parameter in the input/output is small

Motivation (Parameterized Algorithms)

- Faster algorithms for np-hard problems
- Assuming some parameter in the input/output is small
- Allow Runtime to have any computable function of that parameter (as it can be considered constant)

Motivation (Parameterized Algorithms)

- Faster algorithms for np-hard problems
- Assuming some parameter in the input/output is small
- Allow Runtime to have any computable function of that parameter (as it can be considered constant)
- Many tools such as branching, iterative compression, kernelization, tree-decomposition, etc. introduced in the book "Parameterized algorithms"

Motivation (Parameterized Algorithms)

- Faster algorithms for np-hard problems
- Assuming some parameter in the input/output is small
- Allow Runtime to have any computable function of that parameter (as it can be considered constant)
- Many tools such as branching, iterative compression, kernelization, tree-decomposition, etc. introduced in the book "Parameterized algorithms"
- We will focus on tree-decomposition in this thesis

Objectives

- Study parameterized algorithms

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem
- Study Computational Algebraic Geometry

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem
- Study Computational Algebraic Geometry
- Find new parameterized algorithms for the Prime Implicant problem

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem
- Study Computational Algebraic Geometry
- Find new parameterized algorithms for the Prime Implicant problem
- Find new hardness results for the prime implicant problem

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem
- Study Computational Algebraic Geometry
- Find new parameterized algorithms for the Prime Implicant problem
- Find new hardness results for the prime implicant problem
- Find new results related to the Grobner basis problem, if possible

Objectives

- Study parameterized algorithms
- Study the contents related to the Prime Implicant problem
- Study Computational Algebraic Geometry
- Find new parameterized algorithms for the Prime Implicant problem
- Find new hardness results for the prime implicant problem
- Find new results related to the Grobner basis problem, if possible
- Find relationship between the two problems

Table of Contents

Definitions: Parameterized Algorithm

- Given a finite alphabet Σ, any language $L \subseteq \Sigma^* \times \mathbb{N}$ is a parameterized algorithm. And given an instance $(x, k) \in L$, k is the parameter

Definitions: Parameterized Algorithm

- Given a finite alphabet Σ, any language $L \subseteq \Sigma^* \times \mathbb{N}$ is a parameterized algorithm. And given an instance $(x, k) \in L$, k is the parameter
- For practical application, we assume that k is very small (constant) compared to the size of x.

Definitions: Parameterized Algorithm

- Given a finite alphabet Σ, any language $L \subseteq \Sigma^* \times \mathbb{N}$ is a parameterized algorithm. And given an instance $(x, k) \in L$, k is the parameter
- For practical application, we assume that k is very small (constant) compared to the size of x.
- This is restricting the structure of the problem (assuming some parameter of the input/output is bounded)

Definitions: Parameterized Algorithm

- Given a finite alphabet Σ, any language $L \subseteq \Sigma^* \times \mathbb{N}$ is a parameterized algorithm. And given an instance $(x, k) \in L$, k is the parameter
- For practical application, we assume that k is very small (constant) compared to the size of x.
- This is restricting the structure of the problem (assuming some parameter of the input/output is bounded)
- So, parameterization can happen in two ways: input or output. We can also parameterize by multiple parameters.

Runtime

- Given an instance $(x, k) \in L$, L is FPT if there is an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that given any $(x, k) \in \Sigma^* \times \mathbb{N}$, $\mathcal{A}$ can decide if $(x, k) \in L$ in time at most $f(k) \cdot |(x, k)|^c$

Runtime

- Given an instance $(x, k) \in L$, L is FPT if there is an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that given any $(x, k) \in \Sigma^* \times \mathbb{N}$, $\mathcal{A}$ can decide if $(x, k) \in L$ in time at most $f(k) \cdot |(x, k)|^c$
- It is XP, if there is an algorithm $\mathcal{A}$ and two computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that given any $(x, k) \in \Sigma^* \times \mathbb{N}$, $\mathcal{A}$ can decide if $(x, k) \in L$ in time at most $f(k) \cdot |(x, k)|^{g(k)}$

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k
- Given a graph G and k, we check whether the graph has a vertex cover of size k and if yes, we output one vertex cover.

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k
- Given a graph G and k, we check whether the graph has a vertex cover of size k and if yes, we output one vertex cover.
- A brute force algorithm is to check for all k-subsets of the vertex set whether it is a vertex cover

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k
- Given a graph G and k, we check whether the graph has a vertex cover of size k and if yes, we output one vertex cover.
- A brute force algorithm is to check for all k-subsets of the vertex set whether it is a vertex cover
- checking if a subset is a vertex cover can by done in polynomial time, say $poly(|\text{input}|)$

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k
- Given a graph G and k, we check whether the graph has a vertex cover of size k and if yes, we output one vertex cover.
- A brute force algorithm is to check for all k-subsets of the vertex set whether it is a vertex cover
- checking if a subset is a vertex cover can by done in polynomial time, say $poly(|\text{input}|)$
- Then total runtime is $n^k \cdot poly(|\text{input}|)$

Example-1: XP algorithm for vertex cover

- vertex cover problem parameterized by output size k
- Given a graph G and k, we check whether the graph has a vertex cover of size k and if yes, we output one vertex cover.
- A brute force algorithm is to check for all k-subsets of the vertex set whether it is a vertex cover
- checking if a subset is a vertex cover can by done in polynomial time, say $poly(|\text{input}|)$
- Then total runtime is $n^k \cdot poly(|\text{input}|)$
- If k is constant (say ≤ 10), then this is essentially a polynomial time algorithm

Example-2: FPT algorithm for vertex cover

- vertex cover problem parameterized output size again

Example-2: FPT algorithm for vertex cover

- vertex cover problem parameterized output size again
- We can use a branching algorithm where for each vertex v we branch on whether to take it in output or no

Example-2: FPT algorithm for vertex cover

- vertex cover problem parameterized output size again
- We can use a branching algorithm where for each vertex v we branch on whether to take it in output or no
- If we do not take v, then we must take each of its neighbor in the output. So, if it has more than k neighbors, we must take v.

Example-2: FPT algorithm for vertex cover

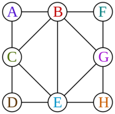
- vertex cover problem parameterized output size again
- We can use a branching algorithm where for each vertex v we branch on whether to take it in output or no
- If we do not take v, then we must take each of its neighbor in the output. So, if it has more than k neighbors, we must take v.
- If at any point we have more than k vertices in our output set, we back-track

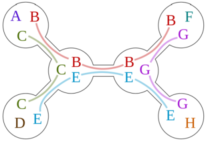
Example-2: FPT algorithm for vertex cover

- vertex cover problem parameterized output size again
- We can use a branching algorithm where for each vertex v we branch on whether to take it in output or no
- If we do not take v, then we must take each of its neighbor in the output. So, if it has more than k neighbors, we must take v.
- If at any point we have more than k vertices in our output set, we back-track
- Note that the branching tree has depth at most k and each node has 2 children (whether to take that node or no). So, runtime is $2^k \cdot poly(|\text{input}|)$ for some polynomial $poly$

Tree-decomposition

- Given a graph $G = (V, E)$, a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ (where T is a tree and $X_t \subseteq V(G)$ for each t) is a tree decomposition of G if
  1. $\bigcup_{t \in V(T)} X_t = V(G)$
  2. $\forall uv \in E(G)$, there is $t \in V(T)$ such that $u, v \in X_t$.
  3. For every $u \in V(G)$, the set $T_u = \{t \in V(T) \ : \ u \in X_t\}$ is a connected sub-tree of T.





Tree-width

- Given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G, its width is $\max_{t \in T} |X_t| - 1$

Tree-width

- Given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G, its width is $\max_{t \in T} |X_t| - 1$
- Tree-width, $tw(G)$, is minimum such width of a tree decomposition

Tree-width

- Given a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G, its width is $\max_{t \in T} |X_t| - 1$
- Tree-width, $tw(G)$, is minimum such width of a tree decomposition
- This is capturing how closely G resembles a tree

Application of Tree-width

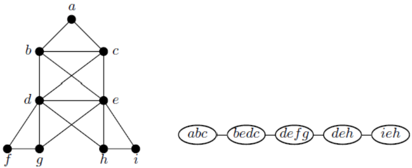
- Many np-hard problems become FPT when parameterized by tree-width

Application of Tree-width

- Many np-hard problems become FPT when parameterized by tree-width
- For solving system of linear equations, the runtime improves from cubic to linear when parameterized by the tree-width of the primal graph

Path-width

- one can define path-width $pw(G)$, where instead of having a tree-decomposition, we have path-decomposition (i.e., we require T to be a path instead of a tree)



Tree-depth

- The tree-depth of a graph G is defined as the smallest possible height of a forest F, where for each edge (u, v) in G, one of u, v is the ancestor of the other in F

Tree-depth

- The tree-depth of a graph G is defined as the smallest possible height of a forest F, where for each edge (u, v) in G, one of u, v is the ancestor of the other in F
- Alternatively, it is defined as follows:

$$td(G) = \begin{cases} 1, & \text{if } |G| = 1 \\ 1 + \min_{v \in V} td(G - v), & \text{if } G \text{ is connected and } |G| > 1 \\ \max_i td(G_i), & \text{Otherwise} \end{cases}$$

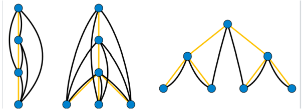


Figure: Black edges are from original graph and yellow ones are from F

Tree-depth

- The tree-depth of a graph G is defined as the smallest possible height of a forest F, where for each edge (u, v) in G, one of u, v is the ancestor of the other in F
- Alternatively, it is defined as follows:

$$td(G) = \begin{cases} 1, & \text{if } |G| = 1 \\ 1 + \min_{v \in V} td(G - v), & \text{if } G \text{ is connected and } |G| > 1 \\ \max_i td(G_i), & \text{Otherwise} \end{cases}$$

- It is well-known that $tw(G) \leq pw(G) \leq td(G) - 1 \leq tw(G) \log n - 1$ for any graph G, where n is number of vertices.

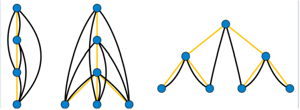


Figure: Black edges are from original graph and yellow ones are from F

Table of Contents

Definitions: Prime Implicant

- A literal is a boolean variable (x) or its negation ($\neg x$)

Definitions: Prime Implicant

- A literal is a boolean variable (x) or its negation ($\neg x$)
- A conjunct is a conjunction of several literals (e.g., $l_1 \wedge l_2 \wedge \cdots \wedge l_k$)

Definitions: Prime Implicant

- A literal is a boolean variable (x) or its negation ($\neg x$)
- A conjunct is a conjunction of several literals (e.g., $l_1 \wedge l_2 \wedge \cdots \wedge l_k$)
- A boolean expression is in disjunctive normal form (d.n.f), if it is a disjunction of several conjuncts (e.g., $c_1 \vee c_2 \vee \cdots \vee c_m$, where $c_1, \cdots, c_m$ are conjuncts)

Definitions: Prime Implicant

- A literal is a boolean variable (x) or its negation ($\neg x$)
- A conjunct is a conjunction of several literals (e.g., $l_1 \wedge l_2 \wedge \cdots \wedge l_k$)
- A boolean expression is in disjunctive normal form (d.n.f), if it is a disjunction of several conjuncts (e.g., $c_1 \vee c_2 \vee \cdots \vee c_m$, where $c_1, \cdots, c_m$ are conjuncts)
- An implicant of a boolean expression is a conjunct that implies the expression (i.e., if conjunct $c \implies b$, where b is any boolean expression, then c is an implicant of b)

Definitions: Prime Implicant

- A literal is a boolean variable (x) or its negation ($\neg x$)
- A conjunct is a conjunction of several literals (e.g., $l_1 \wedge l_2 \wedge \cdots \wedge l_k$)
- A boolean expression is in disjunctive normal form (d.n.f), if it is a disjunction of several conjuncts (e.g., $c_1 \vee c_2 \vee \cdots \vee c_m$, where $c_1, \cdots, c_m$ are conjuncts)
- An implicant of a boolean expression is a conjunct that implies the expression (i.e., if conjunct $c \implies b$, where b is any boolean expression, then c is an implicant of b)
- For example, a is an implicant of $(a \wedge b) \vee (a \wedge \neg b)$

Definitions: Prime Implicant (Cont'd)

- A conjunct c is a prime implicant of a boolean expression b, if c is an implicant of b and for every conjunct c' whose literals are also literals of c (i.e., c' is a "subset" of c), c' is not an implicant of b (i.e., c is a "minimal" implicant of b)

Definitions: Prime Implicant (Cont'd)

- A conjunct c is a prime implicant of a boolean expression b, if c is an implicant of b and for every conjunct c' whose literals are also literals of c (i.e., c' is a "subset" of c), c' is not an implicant of b (i.e., c is a "minimal" implicant of b)
- For example $a \wedge b$ is an implicant of a but not prime implicant.

Definitions: Prime Implicant (Cont'd)

- A conjunct c is a prime implicant of a boolean expression b, if c is an implicant of b and for every conjunct c' whose literals are also literals of c (i.e., c' is a "subset" of c), c' is not an implicant of b (i.e., c is a "minimal" implicant of b)
- For example $a \wedge b$ is an implicant of a but not prime implicant.
- we can similarly define disjunct (disjunction of several literals) and conjunctive normal form c.n.f. (conjunction of several disjuncts)

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants
- The motivation for this is that prime implicants are minimal implicants

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants
- The motivation for this is that prime implicants are minimal implicants
- i.e. no subset of a prime implicant is another implicant

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants
- The motivation for this is that prime implicants are minimal implicants
- i.e. no subset of a prime implicant is another implicant
- So, if we have the set of all prime implicants, we can choose a minimal subset of it and write an equivalent d.n.f. expression

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants
- The motivation for this is that prime implicants are minimal implicants
- i.e. no subset of a prime implicant is another implicant
- So, if we have the set of all prime implicants, we can choose a minimal subset of it and write an equivalent d.n.f. expression
- This d.n.f expression is minimal in the sense that if we remove one conjunct from it or if we remove some literals from a conjunct, the expression won't be equivalent anymore

Problem Definition

- Given a boolean expression in c.n.f. form, we want to compute all its prime implicants
- The motivation for this is that prime implicants are minimal implicants
- i.e. no subset of a prime implicant is another implicant
- So, if we have the set of all prime implicants, we can choose a minimal subset of it and write an equivalent d.n.f. expression
- This d.n.f expression is minimal in the sense that if we remove one conjunct from it or if we remove some literals from a conjunct, the expression won't be equivalent anymore
- We will parameterize the primal graph: vertices are variables and two vertices are connected if they appear in the same disjunct (for c.n.f.) or conjunct (for d.n.f.).

Example

- consider the expression $(a \wedge b) \vee (a \wedge \neg b)$

Example

- consider the expression $(a \wedge b) \vee (a \wedge \neg b)$
- We can find that its only prime implicant is a

Example

- consider the expression $(a \wedge b) \vee (a \wedge \neg b)$
- We can find that its only prime implicant is a
- So, it can be simplified to a.

Initial Results

- **Lemma 1:** Given a c.n.f. boolean expression b, it can be checked in polynomial time if $b \equiv TRUE$.

Initial Results

- **Lemma 1:** Given a c.n.f. boolean expression b, it can be checked in polynomial time if $b \equiv TRUE$.
- **Lemma 2:** Given a conjunct c and a c.n.f. b, it can be checked in polynomial time if c is a prime implicant of b.

Algorithm for Lemma-2

The following is an algorithm to check if conjunct c is an implicant of c.n.f. b.

Algorithm Sketch.

- Substitute truth values according to b in c

□

Algorithm for Lemma-2

The following is an algorithm to check if conjunct c is an implicant of c.n.f. b.

Algorithm Sketch.

- Substitute truth values according to b in c
- Simplify b: if a disjunct containts true remove that; if it contains false, remove the false

$\square$

Algorithm for Lemma-2

The following is an algorithm to check if conjunct c is an implicant of c.n.f. b.

Algorithm Sketch.

- Substitute truth values according to b in c
- Simplify b: if a disjunct containts true remove that; if it contains false, remove the false
- Use the algorithm from Lemma-1 to check if the remaining expression is true.

□

Algorithm for Lemma-2 (Cont'd)

The following is an algorithm to check if conjunct c is a prime implicant of c.n.f. b.

Algorithm Sketch.

- for each conjunct, which is obtained by removing one literal from b, check if it is an implicant (using previous slide's algorithm)

□

Algorithm for Lemma-2 (Cont'd)

The following is an algorithm to check if conjunct c is a prime implicant of c.n.f. b.

Algorithm Sketch.

- for each conjunct, which is obtained by removing one literal from b, check if it is an implicant (using previous slide's algorithm)
- If any of them is an implicant, then output *FALSE*

$\square$

Algorithm for Lemma-2 (Cont'd)

The following is an algorithm to check if conjunct c is a prime implicant of c.n.f. b.

Algorithm Sketch.

- for each conjunct, which is obtained by removing one literal from b, check if it is an implicant (using previous slide's algorithm)
- If any of them is an implicant, then output *FALSE*
- check if b is an implicant. If so, output *TRUE*, otherwise *FALSE*

$\square$

Parameterized by Vertex Cover

- We present an FPT algorithm in the following slide

Parameterized by Vertex Cover

- We present an FPT algorithm in the following slide
- The runtime is $3^k \times poly(\text{input size})$ for some polynomial *poly*, where k is the size of the smallest vertex cover.

Parameterized by Vertex Cover (Example)

Let $b = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_5)$ Note that

- $VC = \{x_1, x_2\}$ is a vertex cover
- other vertices form an independent set (follows from definition of vertex cover)
- means that each of x_3, x_4, x_5 appear alone in disjuncts.

Now suppose $VC' = \{x_1\}$, and $c = \neg x_1$ Then $b' = (x_2 \vee x_3) \wedge (\neg x_2 \vee x_4)$. Then $\neg x_1 \wedge x_3 \wedge x_4$ is an implicant as if x_3 and x_4 are true, then $b' = \mathit{true}$. Thus it is added to the set PI.

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations
- Now, suppose c is a conjunct made from subset VC'

□

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations
- Now, suppose c is a conjunct made from subset VC'
- We assign c in b and are left with c.n.f. b'

□

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations
- Now, suppose c is a conjunct made from subset VC'
- We assign c in b and are left with c.n.f. b'
- One can show that all variables of $V \setminus VC$ will appear alone in disjuncts and there should be no disjunct with variables only from $VC \setminus VC'$

□

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations
- Now, suppose c is a conjunct made from subset VC'
- We assign c in b and are left with c.n.f. b'
- One can show that all variables of $V \setminus VC$ will appear alone in disjuncts and there should be no disjunct with variables only from $VC \setminus VC'$
- Remove those variables from b', so $c \wedge b'$ will be an implicant

□

Parameterized by Vertex Cover (Algorithm Sketch)

Sketch Of Algorithm.

- Given boolean b, let VC be a vertex cover in the primal graph with size k
- We iterate through all subsets of it and for each subset, all possible truth assignments of it. So, we have 3^k iterations
- Now, suppose c is a conjunct made from subset VC'
- We assign c in b and are left with c.n.f. b'
- One can show that all variables of $V \setminus VC$ will appear alone in disjuncts and there should be no disjunct with variables only from $VC \setminus VC'$
- Remove those variables from b', so $c \wedge b'$ will be an implicant
- This gives a set of 3^k implicants. We use previous algorithm to check which of those is a prime implicant

□

Parameterized by Maximum Size of a Prime Implicant

Sketch of an XP algorithm.

- For any subset of the variables of size at most k, assign all possible to truth assignment to these variables

$\square$

Parameterized by Maximum Size of a Prime Implicant

Sketch of an XP algorithm.

- For any subset of the variables of size at most k, assign all possible to truth assignment to these variables
- Use the algorithm from lemma 2 to check if this conjunct is a prime implicant

$\square$

Parameterized by Maximum Size of a Prime Implicant

Sketch of an XP algorithm.

- For any subset of the variables of size at most k, assign all possible to truth assignment to these variables
- Use the algorithm from lemma 2 to check if this conjunct is a prime implicant
- Overall runtime is $\mathcal{O}(n^k \cdot 2^k \cdot poly(n, k))$ where $poly(n, k)$ is some polynomial over n and k, where k is the Maximum Size of a Prime Implicant

$\square$

Parameterized by Number of Disjuncts

- XP algorithm (Lemma 4)

Parameterized by Number of Disjuncts

- XP algorithm (Lemma 4)
- We can show that this is a special case of the previous parameterization

Proof Sketch.

Suppose k is the number of disjuncts. We can show that number of literals in a prime implicant is bounded by k. Any prime implicant intersects with each disjunct in at least one literal and it cannot contain anything extra. $\square$

Hardness Result: Tree-depth and Max Degree

Lemma 5: Number of prime implicants can be exponential even when tree-depth or max degree is bounded.

Proof Sketch.

Consider the example: $b = (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_{n-1} \vee x_n)$ Then b has $2^{\frac{n}{2}}$ prime implicants. Note that tree-depth is 2 and max degree, tree-width, path-width are 1. $\square$

Hardness Result: Parameterized by number of disjuncts

Lemma 6: The problem is XP or worse

Proof Sketch.

Let $b = (x_1 \vee x_2 \vee \cdots \vee x_{\frac{n}{2}}) \wedge (x_{\frac{n}{2}+1} \vee \cdots \vee x_n)$ It has $\left(\frac{n}{2}\right)^2$ prime implicants. Replace 2 with K to get $\Omega(n^k)$ prime implicants. $\square$

Hardness Result: Parameterized by Vertex Cover for d.n.f. input

Lemma 7: The problem is XP or worse

Proof Sketch.

We can construct an example[a] where number of prime implicants is $\left(\frac{n}{k}\right)^k$, where k is the size of a minimum vertex cover. $\square$

[a] please see my final report

Table of Contents

Summary

the results obtained in this thesis can be grouped into three main categories

- novel parameterized algorithms for computing all prime implicants of a given c.n.f. boolean expression.
    - we developed an FPT algorithm parameterized by the vertex cover of the primal graph
    - an XP algorithm parameterized by the maximum size of a prime implicant
    - an XP algorithm parameterized by the number of disjuncts

Summary

the results obtained in this thesis can be grouped into three main categories

- novel parameterized algorithms for computing all prime implicants of a given c.n.f. boolean expression.
  - we developed an FPT algorithm parameterized by the vertex cover of the primal graph
  - an XP algorithm parameterized by the maximum size of a prime implicant
  - an XP algorithm parameterized by the number of disjuncts
- we established hardness results for several other parameters for the same problem.
  - NP-hard even when one of the parameters tree-width, tree-depth, path-width, and degree is bounded
  - XP or worse when parameterized by number of disjuncts
  - XP or worse when the smallest vertex cover Size is bounded

Future Ideas

- Computing all prime implicants for a given c.n.f. in time that is a linear function of input + output size.

Q&A